基本概念:

- 在儲存資料的時候我們有一個value1 跟value2 前者可以把它想成標籤的概念
  而後者則為我們要儲存的內容
  例如:
  學生的ID以及學生的基本資料
- key:
  當我們有了value1 之後我們會用一個 hash function 把value 1轉換成 key
  這邊的key 就如同array內的index一樣直接告訴我們儲存的位置

Hash function:

- 一個好的hash func 需要
  1. Deterministic 當我們有same value1 要轉換成same key
  2. uniform 分散均勻
  3. efficient 有效率

Collision :

- 當我們資料變多的時候勢必會發生不同的value1 轉換變成同一個key
  這是就稱之為coliisoion

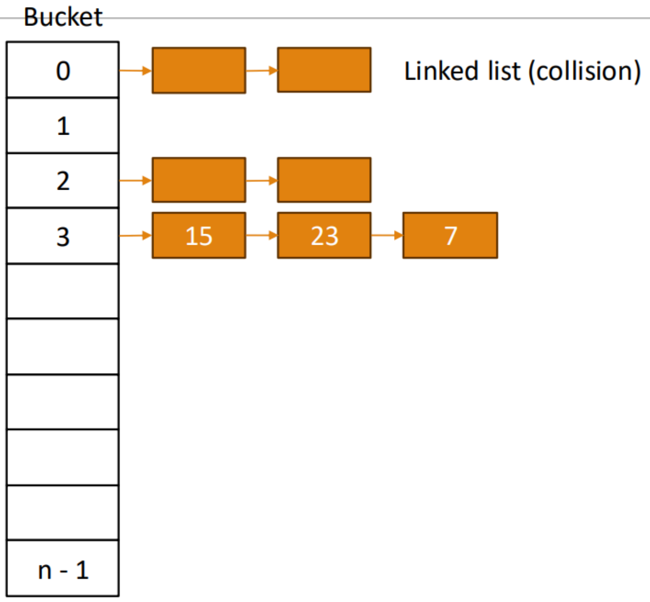
- handling:
  1. chaining : 我們在每格bucket都用linked list去儲存
  2. open addressing 當我們遇到相同key 時往下去找還未被放入的bucket
  3. composite key 因為只用value1太容易造成相同的value1出現例如兩個人都叫做bob 但我們再加入 它們的vlaue2 生日的話就會有不同的value去轉換，總之就次讓value1變得更多樣
  4. Hash Refinement 重寫hash func

常見的hash func:

Hash Function

Method	Formula / Idea	Example
Division Method	`h(k) = k mod m`	key = 123, m = 10 → index = 3
Multiplication Method	`h(k) = floor(m * (k*A mod 1))`, $0 < A < 1$	$A \approx 0.618$
Folding Method	Split key into parts and add them	Key = 123456 → 12+34+56=102
String Hashing	Polynomial rolling hash	`h(s) = (Σ s[i] * p^i) mod m`

Hash Table



Hash(key) = key mod n
0, …, n-1

open addressing:
probing(探測):

What is Probing?

Probing is a **collision-resolution technique** used in **open addressing** hash tables.

When two or more keys map to the same hash index (collision), *probing* defines how the algorithm searches for the **next available slot** in the table.

Probing = systematic search for an empty position in a hash table after a collision.

Typing of Probing

Method	Formula	Behavior	Pros / Cons
Linear Probing	$(h(k) + i) \bmod m$	Check next slot sequentially.	✅ Simple ❌ Primary clustering
Quadratic Probing	$(h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$	Gaps grow quadratically.	✅ Reduces clustering ❌ May skip slots
Double Hashing	$(h_1(k) + i \cdot h_2(k)) \bmod m$	Uses a 2nd hash for step size.	✅ Better spread ❌ More computation

i = probe sequence index (0, 1, 2, …)
hash function: $h(k)$, $h_1(k)$, $h_2(k)$
m: table size

Key Property

Property	Description
Deterministic	Same key always probes same sequence.
Bounded	Will examine at most *m* slots.
Cluster Formation	Some probing methods (e.g. linear) create contiguous filled regions, slowing performance.
Load Factor Sensitivity	As load factor (α = n/m) increases, probe lengths and time complexity rise rapidly.

bounded 最多找m個slot m為 size

cluster叢集如果用linear probe的話 就會造成連續的資料放入cluster越變越長時間複雜度上升

=>Primary Clustering

Linear probing 會造成一段連續 **filled slots**（**cluster**），而新 **key** 又會被迫接在 **cluster** 後面，導致 **cluster** 越變越長，整體速度急遽下降。

Secondary Clustering不同於prmary 只要塞到那個區段就會繼續變長
secondary是當h(key)結果相同時會跑同樣的probe sequence例如

📌 **超直覺例子（Quadratic probing）**

23, 33, 43 都：

makefile　　Copy code

```makefile
h(k) = 3
i = 0 → 3
i = 1 → 3 + 1 + 1 = 5
i = 2 → 3 + 2 + 4 = 9
```

他們會去 **3 → 5 → 9**
（固定 pattern）

但如果 key hash 到 4 就完全走另一組序列
→ 不會跟 hash=3 那組混在一起
→ cluster 很小、不會造成大規模連續塞車

Double Hashing

$h_1(k) = k \bmod 10$

$h_2(k) = 7 - (k \bmod 7)$

Collision: $index(i) = (h_1(k) + i \times h_2(k)) \bmod 10$

Insert keys: 23, 33, 43

Key	$h_1(k)$	i	$h_2(k)$	index(i)	slot	Slot Status
23	3	0	7 - (23 mod 7) = 7 – 2 = 5	3	3	slot[3] = 23
33	3	0	7 – (33 mod 7) = 7 – 5 = 2	3	3	slot[3] = 23, occupied
33	3	1	7 – (33 mod 7) = 7 – 5 = 2	5	5	slot[5] = 33
43	3	0	7 – (43 mod 7) = 7 – 1 = 6	3	3	slot[3] = 23, occupied
43	3	1	7 – (43 mod 7) = 7 – 1 = 6	9	9	slot[9] = 43

Observation:

Jump size, well-distributed across table. Low clustering and better performance for high load factors.

11402 CS203A, COMPUTER SCIENCE & ENGINEERING, YUAN ZE UNIVERSITY

當算出來的index已經被occupied那此時i+=1繼續算

Design hash func:

- mod
- Folding Method其實就是把大數字拆成小數字
  eg Key = 123456 → 12+34+56=102 h(123456) = 102 mod 10 = 2

- non-integer ASCII
- polynominal rolling hash

Weighted String Hash (better spread); Polynomial rolling hash

$h(s) = (\Sigma\, s[i] * p^i) \bmod m$

$h(\text{"CAT"}) = (C\times 31^2 + A\times 31^1 + T\times 31^0) \bmod m$

Time Complexity

Separate Chaining

Operation	Best	Average	Worst	Remarks
Search	O(1)	$O(1 + \alpha)$	O(n)	Average-case constant if α small
Insert	O(1)	O(1)	O(n)	Append to short chain
Delete	O(1)	O(1)	O(n)	Search + unlink node

$T_{avg} \approx O(1+n/m)=O(1+\alpha)$

alpha=n/m =平均每個 bucket 的鏈結串列長
→ 搜尋時需要掃描 α 個節點
→ 加上 hash 計算 O(1)
→ 總共 $O(1 + \alpha)$

Time Complexity

Open Addressing

- Collisions resolved by probing (linear, quadratic, or double hashing).

Operation	Average ($\alpha \le 0.7$)	Worst	Notes
Search	O(1)	O(n)	At high load factor, probe chain length ↑
Insert	O(1)	O(n)	May require several probes
Delete	O(1)	O(n)	Needs careful slot marking (“lazy delete”)

ADT: Dictionary

ADT *Dictionary* is

objects:

A collection of n > 0 pairs, each pair has a key and an associated item

functions:

for all *d* ∈ *Dictionary*, *item* ∈ *Item*, *k* ∈ *Key*, *n* ∈ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(*d*, *n*)	::=	**if** (*n* > 0) **return** TRUE **else return** FALSE
Element Search(*d*, *k*)	::=	**return** *item* with key *k*. **return** NULL if no such element.
Element Delete(*d*, *k*)	::=	delete and return item (if any) with key *k*.
void Insert(d, item, k)	::=	insert *item* with key *k* into *d*.

end *Dictionary*

ADT: HashTable with Separate Chaining

ADT *HashTable* is

objects:

A finite set of pairs <key, value> where key is unique. Keys are distributed across m buckets using hash function h: key → [0, m-1]. Each bucket contains a chain (linked list) of key-value pairs.

parameters:

m: number of buckets (positive integer)

h: hash function (deterministic, uniform distribution)

λ: load factor = n/m where n = number of stored pairs

MAX_LOAD_FACTOR: threshold for triggering resize (default: 0.75)

functions:

for all *h* ∈ *HashTable*, *k* ∈ *Key*, *v* ∈ *Value*

HashTable Create(*m*)	::=	precondition: m > 0 postcondition: return empty hash table with m buckets, λ = 0
Boolean IsEmpty(*h*)	::=	**return** (size(*h*) == 0)
Insert(h, k, v)	::=	i = *h*(*k*) mod m if *k* exists in bucket[i]: replace existing value with v else: add <*k*, *v*> to front of bucket[*i*], increment size if λ > MAX_LOAD_FACTOR: resize(H, 2*m)
value Retrieve(h, k)	::=	i = h(k) mod m search bucket[i] for key k if found: return associated value **else** throw KeyNotFoundException
Boolean Delete(h, k)	::=	i = h(k) mod m if k exists in bucket[i]: remove <k,v>, decrement size, **return** TRUE **else return** false
Boolean Search(h, k)	::=	i = h(k) mod m **return** (k exists in bucket[i])
Iterator Traverse(h)	::=	**return** iterator that visits all key-value pairs order: bucket[0] to bucket[m-1], within bucket: insertion order

end *HashTable*